Summary

I think we should:

- Merge datamodel and specification
- Create a monorepo with dina-collections-ui, dina-style, dina-schema... (if we use node include it here)

I think we could:

- If Mikko, Ida and Ingimar are excited about it, use Node instead of Java for the API

Content below:

1. Current workflow
2. Issues related to current workflow
3. Merge datamodel and specification
4. Create monorepo
5. Migrate to node
6. Manage database migrations (Ingimar raised this in a mail)

1. Current workflow

This is a rough overview of the workflow (based on the last sprint):

- Sprint planning - mainly Mikko + Stefan that creates sprint goals
- Lisa and Markus iterate data model and register form specification
- Lisa and Markus add and change the temporary register form specification in google spreadsheet
- Lisa and Marcus updated and maintain external data model wiki documentation
- Anton and Fredrik discuss register form specification/data model with Marcus, Lisa and Mikko
- Anton and Fredrik (mainly Fredrik) implements changes in dina-schema and in the register form frontend
- Anton and Fredrik communicate with Ida about changes in API (referring to updated dina-schema)
- Ida discuss with Anton, Fredrik, Lisa and Marcus about changed needed to database and implement API endpoints
- Ida updated database + api
- Ingimar does infrastructure

2. Issues related to current tech stack and workflow

What we have now is a good start but there are some issues:

- Lot of work to keep everything in sync: If Lisa and Marcus want to update something in the datamodel related to the register form they need to update the external documentation and update the spreadsheet, Anton/Fredrik need to update the schema and update the frontend, Ida need to update the API and the database schema.
- Confusing and unnecessary that the data model and api have different namings. Adds unclarity when talking about concepts.
- Hard to keep track of stuff being in sync. Ex how do we ensure that the external data model maps to the api? As it is now its named slightly different and is updated at different times.
- Unclear what the source of truth is and unclear how the datamodel, api-documentation and database related to each other
- Information about what should be changed in the API comes too late to Ida

3. Merge datamodel and specification

I think a lot of the issues could be solved by merging the datamodel and the specification.
In practice this mean:

- Lisa and Marcus work with the models and attributes in the dina-schema (we should likely rename this). Here they can add all attributes for all models, describe models and attribute. Both describing the data and format and also write documenting text explaining what it is and so forth. Here it will also be possible to add example data.
- Devs works with the endpoints in the specification (which uses the models and attributes)
- Based on this we can generate:
    - Api documentation,
    - External documentation (replace the wiki)
    - Validations that we can use in frontend forms and in api communication.
    - Automatically generate tests both for the frontend and the api (example data in spec or automatically generated mock data can be used).
    - We could also make all of this versioned and in the future make it possible to brows different version of documentation and also automatically generate reasonable diffes between different versions.
- We work with git so Lisa and Marcus can work in different branches, both for prototyping and for the next version and when something is ready they can create an MR that Anton, Fredrik, Mikko and Ida can review. In this way communication of changes becomes very clear and failing tests will be a good start for a todo list.
- With this setup this specification will become the single source of truth both for human (internal and external) communication and also for communication between machines. The database tables will be managed by the developers and it will follow from the specification and not the other way around.
- This will help us in full chain interations which i think is one of the core capacities we need to develop now

Clarifying comment: Ingimar is not mentioned as a part of the flow above because he has not yet started coding. Of course he will be included in this.

4. Create a monorepo

With dina-collections-ui, dina-style, dina-schema and dina-collections-docker (if we use node include it here)

The code for creating api-clients that can parse the specification, make test requests and so forth live in dina-collections-ui. This is needed to test the specification and having them live in different repositories makes and versioning harder. Same is true for the styles. Its been a good start to have them separate but now I believe its time to bring them together. If we rewrite the api to node it makes sense to also include the api in this repository. If we keep the api in java I think we should not include it. Even if it would be possible, it makes the deploy, test and Travis flows a bit harder due to different tech and different build and start times. If we later want to include also the api we can do that.

5. Migrate to node

When I talked about this yesterday I mentioned this as an important thing and I think the pros and cons we talked about are relevant:

Pros:

- The team need to know javascript either way because the frontend is written in javascript - having everyone code in javascript will make it much easier to spread knowledge about the frontend.
- Code reusability - we can share a lot of code between the frontend and the backend. Use same validations in the api as in the form, share error-codes. test-data, json schema. A lot of the pieces for the backend already exist in the dina-collections-ui and the schema repo.
- We can fast get to the point where we have proper logging, proper endpoint validation (both for input and output) can create new endpoints that generates mock data from json schema. (I think I would need 1 sprint to get us the the same place in terms of endpoints as we have now but with the above included)
- The api takes 3 seconds to start (not 1 min)
- As we migrate we will use the orm sequalize witch will make it trivial to switch to postgres from mysql (could either be done now or later)
- This will later make it much easier to build layers on top of our application that will make it possible to build stuff from configuration
- Having one repository in one language will improve collaboration between developers. It will be easier to work with code-reviews and spread knowledge
- Me and fredrik will be able to help more with backend coding.

Cons:

- Ingimar and Ida knows and, from what I know, likes Java
- Migrating to node will result in throwing away stuff we (Ida) have build
- There will be a learning curve for ida and Ingimar and Mikko before getting productive.
- Anton will be more involved in the backend, reducing frontend productivity.

Another thing to note is that since we actually can do validation of api-calls already in the client, its not that vital that the api has this validation as long as it doesn't fail on what should be successful calls. If it returns wrong data, the frontend tests will fail, so we can use tests in the frontend to test the backend.

Also, if we decide to not migrate now but do it later that is also fine. It will not take that much time to build it later when we can look at the Java code.

I (Anton) know that I have a bias towards Node, so my recommendation is that Ida, Ingimar and Mikko talk about and make the Node vs Java decision. I think the most important aspect is if Ida, Ingimar and Mikko want to learn and work with JavaScript. If they prefer Java, we should stick with that.

6. Manage database migrations (Ingimar raised this in an email)

Ingimar raised a question regarding data migrations and specifically liquibase in his email. I think that we need that kind of tool when we get closer to production and especially when we have something in production. And if Ingimar has experience working with liquibase is a good choice. If we use node as api there is similar tools (stores migrations in files with version control) in the node-world and might make more sense.

However I think that we should not introduce this now. Now it's important that we can do fast iterations and to keep track of the migrations will slow down iterations. It's easier if we can setup so that the database is always reset every time we deploy and populated with mock data. This is something I think we should invest some more time into building.